Scientific software engineering with a simple ODE model as example

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Mar 25, 2015

Contents

List of Exercises and Problems

Teaching material on scientific computing has traditionally been very focused on the mathematics and the applications, while details on how the computer is programmed to solve the problems have received little attention. Many end up writing as simple programs as possible and are not aware of much useful computer science technology that would increase the fun, efficiency, and reliability of the their scientific computing activities.

This document demonstrates a series of good practices and tools from modern computer science, using a very simple mathematical problem with a very simple implementation such that we minimize the mathematical details. Our goal is to increase the technological quality of computer programming and make it match the more well-established quality of the mathematics of scientific computing.

More specifically we address the following scientific topics:

- How to structure a code in terms of functions
- How to make a module
- How to read input data flexibly from the command line
- How to create graphical/web user interfaces
- How to write unit tests (test functions or doctests)
- How to refactor code in terms of classes (instead of functions only)
- How to conduct and automate large-scale numerical experiments
- How to write scientific reports in various formats (LaTeX, HTML)

The conventions and techniques outlined here will save you a lot of time when you incrementally extend software over time from simpler to more complicated problems. In particular, you will benefit from many good habits:

- new code is added in a modular fashion to a library (modules),
- programs are run through convenient user interfaces,
- it takes one quick command to let all your code can undergo heavy testing,
- tedious manual work with running programs is automated,
- your scientific investigations are reproducible,
- scientific reports with top quality typesetting are produced both for paper and electronic devices.

1 Basic implementations

1.1 Mathematical problem and solution technique

We address the perhaps simplest possible differential equation problem

$$u'(t) = -au(t), \quad t \in (0, T], \tag{1}$$
$$u(0) = I, \tag{2}$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time, e.g., vertical pressure variation in the atmosphere, cooling of an object, and radioactive decay.

The time domain is discretized with points $0 = t_0 < t_1 \cdots < t_{N_t} = T$, here with a constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N_t$. Let u^n be the numerical approximation to the exact solution at t_n. A family of popular numerical methods can be written in the form

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a \Delta t} u^n, \tag{3}$$

for $n = 0, 1, \ldots, N_t - 1$. This numerical scheme corresponds to the Forward Euler scheme when $\theta = 0$, the Backward Euler scheme when $\theta = 1$, and the Crank-Nicolson scheme when $\theta = 1/2$. The initial condition (2) is key to start the recursion with a value for u^0.

1.2 A first, quick implementation

Solving (3) in a program is very straightforward: just make a loop over n and evaluate the formula. The $u(t_n)$ values for discrete n can be stored in an array. This makes it easy to also plot the solution. It would be natural to also add the exact solution curve $u(t) = Ie^{-at}$ to the plot.

Traditionally, a language like Fortran or C would be adopted for implementation, but the enormous popularity of MATLAB in the computational science community over the last two decades, makes a high-level MATLAB-style implementation natural. Below is such a code, written in Python with a syntax close to MATLAB and with a programming style heavenly influenced by "MATLAB scripting". A student or engineer would quickly put together statements like this:

```
from numpy import *
from matplotlib.pyplot import *

A = 1
a = 2
T = 4
dt = 0.2
N = int(round(T/dt))
y = zeros(N+1)
```

```
t = linspace(0, T, N+1)
theta = 1
y[0] = A
for n in range(0, N):
    y[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*y[n]

y_e = A*exp(-a*t) - y
error = y_e - y
E = sqrt(dt*sum(error**2))
print 'Norm of the error: %.3E' % E
plot(t, y, 'r--o')
t_e = linspace(0, T, 1001)
y_e = A*exp(-a*t_e)
plot(t_e, y_e, 'b-')
legend(['numerical, theta=%g' % theta, 'exact'])
xlabel('t')
ylabel('y')
show()
```

This program is easy to read, and as long it is correct, many will claim that it has sufficient quality. Nevertheless, the program suffers from serious flaws that quickly become crucial for writing correct programs for more complicated mathematical problems. First we list two serious bad habits:

1. The notation in the program does not correspond *exactly* to the notation in the mathematical problem: the solution is called `y` and corresponds to u in the mathematical description, the variable `A` corresponds to the mathematical parameter I, `N` in the program is called N_t in the mathematics.

2. There are no comments in the program.

1.3 A more decent, flat program

A code with more satisfactory quality arises from fixing the notation and adding comments:

```
from numpy import *
from matplotlib.pyplot import *

I = 1
a = 2
T = 4
dt = 0.2
Nt = int(round(T/dt))     # no of time intervals
u = zeros(Nt+1)           # array of u[n] values
t = linspace(0, T, Nt+1)  # time mesh
theta = 1                 # Backward Euler method

u[0] = I                  # assign initial condition
for n in range(0, Nt):    # n=0,1,...,Nt-1
    u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]

# Compute norm of the error
u_e = I*exp(-a*t) - u     # exact u at the mesh points
error = u_e - u
```

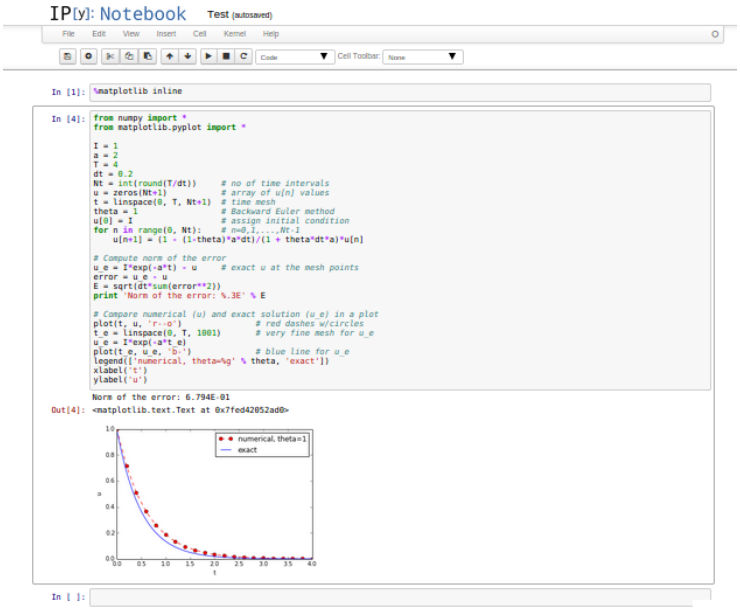


Figure 1: Flat experimental code in a notebook.

```
E = sqrt(dt*sum(error**2))
print 'Norm of the error: %.3E' % E

# Compare numerical (u) and exact solution (u_e) in a plot
plot(t, u, 'r--o')                # red dashes w/circles
t_e = linspace(0, T, 1001)        # very fine mesh for u_e
u_e = I*exp(-a*t_e)
plot(t_e, u_e, 'b-')              # blue line for u_e
legend(['numerical, theta=%g' % theta, 'exact'])
xlabel('t')
ylabel('u')
show()
```

At first sight, this is a good starting point for playing around with the mathematical problem: we can just change parameters and rerun. Let us embed the program in an IPython notebook such that we can get the plot up in the notebook, see Figure 1.

Although such an interactive session is good for initial exploration, one will soon extend the experiments and start developing the code further. Say we want to compare $\theta = 0, 1, 0.5$ in the same plot. This extension requires changes all over the code and quickly lead to errors. To do something serious with this program we have to break it into smaller pieces and make sure each piece is well tested, is general, and can be reused in new contexts without changes. The next natural step is therefore to isolate the numerical computations and the visualization in separate functions.

1.4 Implementing the numerical algorithm in a function

The solution formula (3) is completely general and should be available as a Python function `solver` with all input data as function arguments and all output data returned to the calling code. With this `solver` function we can then solve all types of problems (1)-(2) by an easy-to-read one-line statement:

```
u, t = solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

The implementation of the `solver` function in Python goes like this:

```
def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)                # avoid integer division
    Nt = int(round(T/dt))         # no of time intervals
    T = Nt*dt                     # adjust T to fit time step dt
    u = np.zeros(Nt+1)            # array of u[n] values
    t = np.linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Tip: Always use a doc string to document a function!

Python has a convention for documenting the purpose and usage of a function in a *doc string*: simply place the documentation in a one- or multi-line triple-quoted string right after the function header.

Be careful with unintended integer division!

Note that we in the `solver` function explicitly covert `dt` to a `float` object. If not, the updating formula for `u[n+1]` may evaluate to zero because of integer division when `theta`, `a`, and `dt` are integers!

One of the most serious flaws in computational work is to have several slightly different implementations of the same computational algorithms lying around in various program files. This is very likely to happen, because busy scientists often want to test a slight variation of a code to see what happens. A quick copy and edit do the task, but such quick hacks have a tendency to survive. When a real correction is needed in the implementation, it is difficult to ensure that the correction is done in all relevant files. In fact, this is a general problem in programming, which has led to an important principle.

The DRY principle: Don't repeat yourself!

When implementing a particular functionality in a computer program, make sure this functionality and its variations are implemented in just one piece of code. That is, if you need to revise the implementation, there should be *one and only one* place to edit. It follows that you should never duplicate code (don't repeat yourself!), and code snippets that are similar should be factored into one piece (function) and parameterized (by function arguments).

1.5 Making a module

As soon as you start making Python functions in a program, you should make sure the program file fulfills the requirement of a module. This means that you can import and reuse your functions in other programs too. For example, if our `solver` function resides in a module `decay` in a module file `decay.py`, we can in any program do

```
from decay import solver
u, t = solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

or prefix function names by the module name:

```
import decay
u, t = decay.solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

The requirements for a program to qualify for a module are simple:

1. The filename without `.py` must be a valid Python variable name.
2. The main program must be executed (through statements or a function call) in the *test block*.

The *test block* is normally placed at the end of a module file:

```
if __name__ == '__main__':
    # Statements
```

When the module file is executed as a stand-alone program, the if test is true and the indented statements are run, but when the module file is imported, `__name__` equals the module name and the test block is not executed.

To explain the importance of the test block, consider the trivial module file `hello.py` with one function and a call to this function as main program:

```
def hello(arg='World!'):
    print 'Hello, ' + arg

if __name__ == '__main__':
    hello()
```

Without the test block,

```
def hello(arg='World!'):
    print 'Hello, ' + arg

hello()
```

any attempt to import `hello` will also execute the call `hello()` and hence write 'Hello, World!' to the screen. Such output is not desired when importing a module! However, with the test block, `hello()` is not called during import, but running the file as `python hello.py` will make the block active and lead to the desired printing.

All coming functions are placed in a module.

The many functions to be explained in the following text are put in one module file `decay.py`.

What more than the `solver` function is needed in our `decay` module to do everything we did in the previous, flat program? We need import statements for `numpy` and `matplotlib` as well as another function for producing the plot. It can also be convenient to put the exact solution in a Python function. Our module `decay.py` then looks like this:

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, a, T, dt, theta):
    ...

def exact_solution(t, I, a):
    return I*exp(-a*t)

def experiment_compare_numerical_and_exact():
    I = 1;  a = 2;  T = 4;  dt = 0.4;  theta = 1
    u, t = solver(I, a, T, dt, theta)

    t_e = linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plot(t,   u,   'r--o')           # dashed red line with circles
    plot(t_e, u_e, 'b-')             # blue line for u_e
    legend(['numerical, theta=%g' % theta, 'exact'])
    xlabel('t')
    ylabel('u')
    plotfile = 'tmp'
    savefig(plotfile + '.png');  savefig(plotfile + '.pdf')

    error = exact_solution(t, I, a) - u
    E = sqrt(dt*sum(error**2))
    print 'Error norm:', E

if __name__ == '__main__':
    experiment_compare_numerical_and_exact()
```

This module file does exactly the same as the previous, flat program, but it becomes much easier to extend the code with other plots or experiments in new functions. And even more important, the numerical algorithm is coded and tested once and for all in the `solver` function, and any need to solve the mathematical problem is a matter of one function call.

1.6 Prefixing imported functions by the module name

Import statements of the form `from module import *` import functions and variables in `module.py` into the current file. For example, when doing

```
from numpy import *
from matplotlib.pyplot import *
```

we get mathematical functions like `sin` and `exp` as well as MATLAB-style functions like `linspace` and `plot`, which can be called by these well-known names. Unfortunately, it sometimes becomes confusing to know where a particular function comes from. Is it from `numpy`? Or `matplotlib.pyplot`? Or is it our own function?

An alternative import is

```
import numpy
import matplotlib.pyplot
```

and such imports require functions to be prefixed by the module name, e.g.,

```
t = numpy.linspace(0, T, Nt+1)
u_e = I*numpy.exp(-a*t)
matplotlib.pyplot.plot(t, u_e)
```

This is normally regarded as a better habit because it is explicitly stated from which module a function comes from.

The modules `numpy` and `matplotlib.pyplot` are so frequently used, and their full names quite tedious to write, so two standard abbreviations have evolved in the Python scientific computing community:

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, T, Nt+1)
u_e = I*np.exp(-a*t)
plt.plot(t, u_e)
```

The downside of prefixing functions by the module name is that mathematical expressions like $e^{-at}\sin(2\pi t)$ get cluttered with module names,

```
numpy.exp(-a*t)*numpy.sin(2(numpy.pi*t)
# or
np.exp(-a*t)*np.sin(2*np.pi*t)
```

Such an expression looks like `exp(-a*t)*sin(2*pi*t)` in most other programming languages. Similarly, `np.linspace` and `plt.plot` look less familiar to people who are used to MATLAB and who have not adopted Python's prefix style. Whether to do `from module import *` or `import module` depends on personal taste and the problem at hand. In these writings we use `from module import` in more basic, shorter programs where similarity with MATLAB could be an advantage. Prefix of mathematical functions in formulas is something we often avoid to obtain a one-to-one correspondence between mathematical formulas and the Python code.

Our `decay` module can be edited to use the module prefix for `matplotlib.pyplot` and `numpy`:

```
import numpy as np
import matplotlib.pyplot as plt

def solver(I, a, T, dt, theta):
    ...

def exact_solution(t, I, a):
    return I*np.exp(-a*t)

def experiment_compare_numerical_and_exact():
    I = 1;  a = 2;  T = 4;  dt = 0.4;  theta = 1
    u, t = solver(I, a, T, dt, theta)

    t_e = np.linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plt.plot(t,   u,   'r--o')       # dashed red line with circles
    plt.plot(t_e, u_e, 'b-')         # blue line for u_e
    plt.legend(['numerical, theta=%g' % theta, 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf')

    error = exact_solution(t, I, a) - u
    E = np.sqrt(dt*np.sum(error**2))
    print 'Error norm:', E

if __name__ == '__main__':
    experiment_compare_numerical_and_exact()
```

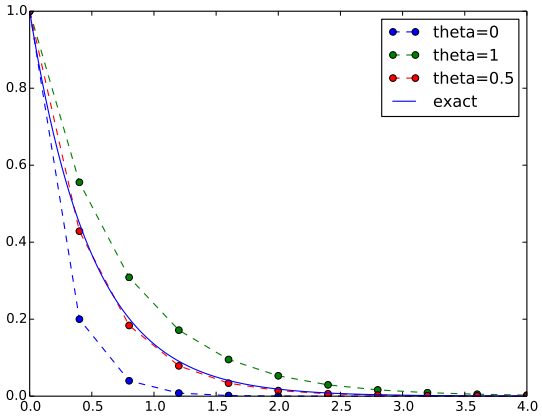
We remark that some would prefer to get rid of the prefix in mathematical formulas:

```
from numpy import exp, sum, sqrt

def exact_solution(t, I, a):
    return I*exp(-a*t)

error = exact_solution(t, I, a) - u
E = sqrt(dt*sum(error**2))
```

1.7 Comparing numerical schemes in one plot

Let us specifically demonstrate one extension of the flat program in Section 1.2 that would require substantial editing of the flat code (Section 1.3), while in a structured module (Section 1.5), we can simply add a new function without affecting the existing code and with reusing the implementation of the numerics.

Our aim is to make a comparison between the numerical solutions for various schemes (θ values) and the exact solution:



Stop a minute!

Look at the flat program in Section 1.2, and try to imagine which edits that are required to solve this new problem.

With the `solver` function at hand, we can simply create a function with a loop over `theta` values and add the necessary plot statements:

```
def experiment_compare_schemes():
    """Compare theta=0,1,0.5 in the same plot."""
    I = 1;  a = 2;  T = 4;  dt = 0.4
    legends = []
    for theta in [0, 1, 0.5]:
        u, t = solver(I, a, T, dt, theta)
        plt.plot(t, u, '--o')            # dashed lines with circles
        legends.append('theta=%g' % theta)
    t_e = np.linspace(0, T, 1001)        # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t_e, u_e, 'b-')             # blue line for u_e
    legends.append('exact')
    plt.legend(legends, loc='upper right')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf')
```

A call to this `experiment_compare_schemes` function must be placed in the test block, or you can run the program from IPython instead:

```
In[1]: from decay import *

In[2]: experiment_compare_schemes()
```

2 User interfaces

It is good programming practice to let programs read input from some *user interface*, rather than requiring users to *edit* parameter values in the source code. With effective user interfaces it becomes easier and safer to apply the code for scientific investigations and in particular to automate large-scale investigations by other programs (see Section 5).

Reading input data can be done in many ways. We have to decide on a the functionality of the user interface, i.e., how we want to operate the program when providing input, and then use appropriate tools to implement the user interface. There are four basic types of user interface, listed here with increasing complexity of the implementation:

1. Questions and answers in the terminal window
2. Command-line arguments
3. Reading data from files
4. Graphical user interfaces (GUIs)

Alternatives 2 and 4 are most popular and will be addressed next. The goal is to make it easy for the user to set physical and numerical parameters in our `decay.py` program.

2.1 Command-line arguments

The command-line arguments are all the words that appear on the command line after the program name. Running a program `prog` as `prog arg1 arg2` means that there are two command-line arguments (separated by white space): `arg1` and `arg2`. Python stores all the command-line arguments in the list `sys.argv`, and there are, in principle, two ways of programming with command-line arguments in Python:

- **Positional arguments:** Decide upon a sequence of parameters on the command line and read their values directly from the `sys.argv[1:]` list.
- **Option-value pairs:** Use `--option value` on the command line to replace the default value of an input parameter `option` by `value` (and utilize the `argparse.ArgumentParser` tool for implementation).

Suppose we want to run some program `prog.py` with specification of two parameters `p` and `delta` on the command line. With positional command-line arguments we write

```
Terminal> python decay.py 2 0.5
```

and must know that the first argument `2` represents `p` and the next `0.5` is the value of `delta`. With option-value pairs we can run

```
Terminal> python decay.py --delta 0.5 --p 2
```

Now, both `p` and `delta` are supposed to have default values in the program, so we need to specify only the parameter that is to be changed from its default value, e.g.,

```
Terminal> python decay.py --p 2         # p=2, default delta
Terminal> python decay.py --delta 0.7   # delta-0.7, default a
Terminal> python decay.py               # default a and delta
```

For our `decay.py` module file, we want include functionality such that we can read I, a, T, θ, and a range of Δt values from the command line. A plot is then to be made, comparing the different numerical solutions for different Δt values against the exact solution. The technical details of getting the command-line information into the program is covered in the next two sections.

2.2 Positional command-line arguments

The simplest way of reading the input parameters is to decide on their sequence on the command line and just index the `sys.argv` list accordingly. Say the sequence is I, a, T, θ followed by an arbitrary number of Δt values. This code extract these *positional* command-line arguments:

```
import sys
I = float(sys.argv[1])
a = float(sys.argv[2])
T = float(sys.argv[3])
theta = float(sys.argv[4])
dt_values = [float(arg) for arg in sys.argv[5:]]
```

Command-line arguments are strings!

Note that all elements in `sys.argv` are string objects. If the values will enter mathematical computations, we need to explicitly convert the strings to numbers.

Instead of specifying the θ value, we could be a bit more sophisticated and let the user write the name of the scheme: `BE` for Backward Euler, `FE` for Forward Euler, and `CN` for Crank-Nicolson. Then we must map this string to the proper θ value, an operation elegantly done by a dictionary:

```
scheme = sys.argv[4]
scheme2theta = {'BE': 1, 'CN': 0.5, 'FE': 0}
if scheme in scheme2theta:
    theta = scheme2theta[scheme]
else:
    print 'Invalid scheme name:', scheme; sys.exit(1)
```

2.3 Option-value pairs on the command line

Now we want to specify option-value pairs on the command line, using `--I` for `I` (I), `--a` for `a` (a), `--T` for `T` (T), `--scheme` for the scheme name (`BE`, `FE`, `CN`), and `--dt` for the sequence of `dt` (Δt) values. Each parameter must have a sensible default value so that we specify the option on the command line only when the default value is not suitable. Here is a typical run:

```
Terminal> python decay.py --I 2.5 --dt 0.1 0.2 0.01 --a 0.4
```

Observe the major advantage over positional command-line arguments: the input is much easier to read and much easier to write. With positional arguments it is easy to mess up the sequence of the input parameters and quite challenging to detect errors too, unless there are just a couple of arguments.

Python's `ArgumentParser` tool in the `argparse` module makes it easy to create a professional command-line interface to any program. The documentation of `ArgumentParser` demonstrates its versatile applications, so we shall here just list an example containing the most basic features. It always pays off to use `ArgumentParser` rather than trying to manually inspect and interpret option-value pairs in `sys.argv`!

The use of `ArgumentParser` typically involves three steps:

```
import argparse
parser = argparse.ArgumentParser()

# Step 1: add arguments
parser.add_argument('--option_name', ...)

# Step 2: interpret the command line
args = parser.parse_args()
```

```
# Step 3: extract values
value = args.option_name
```

A function for setting up all the options is handy:

```
def define_command_line_options():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--I', '--initial_condition', type=float,
        default=1.0, help='initial condition, u(0)',
        metavar='I')
    parser.add_argument(
        '--a', type=float, default=1.0,
        help='coefficient in ODE', metavar='a')
    parser.add_argument(
        '--T', '--stop_time', type=float,
        default=1.0, help='end time of simulation',
        metavar='T')
    parser.add_argument(
        '--scheme', type=str, default='CN',
        help='FE, BE, or CN')
    parser.add_argument(
        '--dt', '--time_step_values', type=float,
        default=[1.0], help='time step values',
        metavar='dt', nargs='+', dest='dt_values')
    return parser
```

Each command-line option is defined through the `parser.add_argument` method. Alternative options, like the short `--I` and the more explaining version `--initial_condition` can be defined. Other arguments are `type` for the Python object type, a default value, and a help string, which gets printed if the command-line argument `-h` or `--help` is included. The `metavar` argument specifies the value associated with the option when the help string is printed. For example, the option for I has this help output:

```
Terminal> python decay.py -h
  ...
  --I I, --initial_condition I
                        initial condition, u(0)
  ...
```

The structure of this output is

```
      --I metavar, --initial_condition metavar
                            help-string
```

Finally, the `--dt` option demonstrates how to allow for more than one value (separated by blanks) through the `nargs='+'` keyword argument. After the command line is parsed, we get an object where the values of the options are stored as attributes. The attribute name is specified by the `dist` keyword argument, which for the `--dt` option is `dt_values`. Without the `dest` argument, the value of an option `--opt` is stored as the attribute `opt`.

The code below demonstrates how to read the command line and extract the values for each option:

```
def read_command_line_argparse():
    parser = define_command_line_options()
    args = parser.parse_args()
    scheme2theta = {'BE': 1, 'CN': 0.5, 'FE': 0}
    data = (args.I, args.a, args.T, scheme2theta[args.scheme],
            args.dt_values)
    return data
```

As seen, the values of the command-line options are available as attributes in `args`: `args.opt` holds the value of option `--opt`, unless we used the `dest` argument (as for `--dt_values`) for specifying the attribute name. The `args.opt` attribute has the object type specified by `type` (`str` by default).

The making of the plot is not dependent on whether we read data from the command line as positional arguments or option-value pairs:

```
def experiment_compare_dt(option_value_pairs=False):
    I, a, T, theta, dt_values = \
       read_command_line_argparse() if option_value_pairs else \
       read_command_line_positional()

    legends = []
    for dt in dt_values:
        u, t = solver(I, a, T, dt, theta)
        plt.plot(t, u)
        legends.append('dt=%g' % dt)
    t_e = np.linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t_e, u_e, '--')            # dashed line for u_e
    legends.append('exact')
    plt.legend(legends, loc='upper right')
    plt.title('theta=%g' % theta)
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf')
```

2.4 Creating a graphical web user interface

The Python package Parampool can be used to automatically generate a web-based *graphical user interface* (GUI) for our simulation program. Although the programming technique dramatically simplifies the efforts to create a GUI, the forthcoming material on equipping our `decay` module with a GUI is quite technical and of significantly less importance than knowing how to make a command-line interface.

Making a compute function. The first step is to identify a function that performs the computations and that takes the necessary input variables as arguments. This is called the *compute function* in Parampool terminology. The purpose of this function is to take values of I, a, T together with a sequence of Δt values and a sequence of θ and plot the numerical against the exact solution

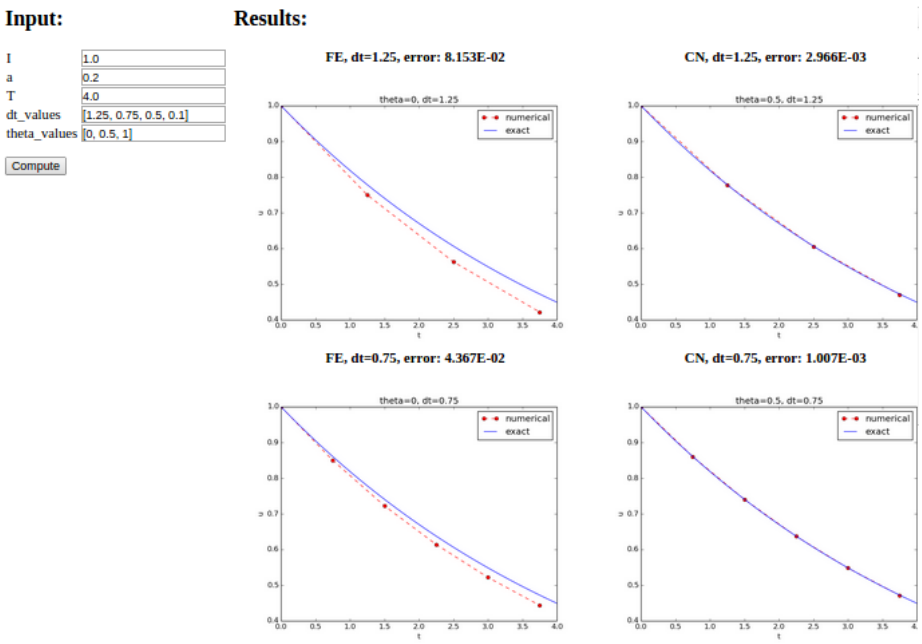


Figure 2: Automatically generated graphical web interface.

for each pair of $(\theta, \Delta t)$. The plots can be arranged as a table with the columns being scheme type (θ value) and the rows reflecting the discretization parameter (Δt value). Figure 2 displays of the graphical web interface may look like after results are computed (there are 3×3 plots in the GUI, but only 2×2 are visible in the figure).

To tell Parampool what type of input data we have, we assign default values of the right type to all arguments in the compute function, here called `main_GUI`:

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
```

The compute function must return the HTML code we want for displaying the result in a web page. Here we want to show a table of plots. Assume for now that the HTML code for one plot and the value of the norm of the error can be computed by some other function `compute4web`. The `main_GUI` function can then loop over Δt and θ values and put each plot in an HTML table. Appropriate code goes like

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
    # Build HTML code for web page. Arrange plots in columns
```

```
    # corresponding to the theta values, with dt down the rows
    theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
    html_text = '<table>\n'
    for dt in dt_values:
        html_text += '<tr>\n'
        for theta in theta_values:
            E, html = compute4web(I, a, T, dt, theta)
            html_text += """
<td>
<center><b>%s, dt=%g, error: %.3E</b></center><br>
%s
</td>
""" % (theta2name[theta], dt, E, html)
        html_text += '</tr>\n'
    html_text += '</table>\n'
    return html_text
```

Making one plot is done in `compute4web`. The statements should be straightforward from earlier examples, but there is one new feature: we use a tool in Parampool to embed the PNG code for a plot file directly in an HTML image tag. The details are hidden from the programmer, who can just rely on relevant HTML code in the string `html_text`. The function looks like

```
def compute4web(I, a, T, dt, theta=0.5):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions in a PNG
    plot whose data are embedded in an HTML image tag.
    """
    u, t = solver(I, a, T, dt, theta)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = np.sqrt(dt*np.sum(e**2))

    plt.figure()
    t_e = np.linspace(0, T, 1001)    # fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t,   u,   'r--o')       # red dashes w/circles
    plt.plot(t_e, u_e, 'b-')         # blue line for exact sol.
    plt.legend(['numerical', 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plt.title('theta=%g, dt=%g' % (theta, dt))
    # Save plot to HTML img tag with PNG code as embedded data
    from parampool.utils import save_png_to_str
    html_text = save_png_to_str(plt, plotwidth=400)

    return E, html_text
```

Generating the user interface. The web GUI is automatically generated by the following code, placed in a file `decay_GUI_generate.py`

```
from parampool.generator.flask import generate
from decay import main_GUI
generate(main_GUI,
         filename_controller='decay_GUI_controller.py',
```

```
            filename_template='decay_GUI_view.py',
            filename_model='decay_GUI_model.py')
```

Running the `decay_GUI_generate.py` program results in three new files whose names are specified in the call to `generate`:

1. `decay_GUI_model.py` defines HTML widgets to be used to set input data in the web interface,
2. `templates/decay_GUI_views.py` defines the layout of the web page,
3. `decay_GUI_controller.py` runs the web application.

We only need to run the last program, and there is no need to look into these files.

Running the web application. The web GUI is started by

```
Terminal> python decay_GUI_controller.py
```

Open a web browser at the location `127.0.0.1:5000`. Input fields for `I`, `a`, `T`, `dt_values`, and `theta_values` are presented. Figure 2 shows a part of the resulting page if we run with the default values for the input parameters. With the techniques demonstrated here, one can easily create a tailored web GUI for a particular type of application and use it to interactively explore physical and numerical effects.

3 Tests for verifying implementations

Any module with functions should have a set of tests that can check the correctness of the implementations. There exists well-established procedures and corresponding tools for automating the execution of such tests. One can in this way, with a one-line command, run large test sets and confirm that the software works (as far as the tests tell!). Here we shall illustrate two important software testing techniques: *doctest* and *unit testing*. The first one is Python specific, while unit testing is the dominating test technique for computer software today.

3.1 Doctests

A doc string, the first string after the function header, is used to document the purpose of functions and their arguments (see Section 1.4). Very often it is instructive to include an example in the doc string on how to use the function. Interactive examples in the Python shell are most illustrative as we can see the output resulting from the statements and expressions. For example, we can in the `solver` function include an example on calling this function and printing the computed `u` and `t` arrays:

```
def solver(I, a, T, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.


    >>> u, t = solver(I=0.8, a=1.2, T=1.5, dt=0.5, theta=0.5)
    >>> for t_n, u_n in zip(t, u):
    ...     print 't=%.1f, u=%.14f' % (t_n, u_n)
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
    """
    ...
```

When such interactive demonstrations are inserted in doc strings, Python's `doctest` module can be used to automate running all commands in interactive sessions and compare new output with the output appearing in the doc string. All we have to do in the current example is to run the module file `decay.py` with

```
Terminal> python -m doctest decay.py
```

This command imports the `doctest` module, which runs all doctests found in the file and reports discrepancies between expected and computed output.

Doctests prevent command-line arguments!

No additional command-line argument is allowed when running doctests. If your program relies on command-line input, make sure the doctests can be run *without* such input.

The execution command above will report any problem if a test fails. For example, changing the last digit `8` in the output of the doctest to `7` triggers a report:

```
Terminal> python -m doctest decay.py
**********************************************************
File "decay.py", line ...
Failed example:
    for t_n, u_n in zip(t, u):
        print 't=%.1f, u=%.14f' % (t_n, u_n)
Expected:
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761947
Got:
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
```

Pay attention to the number of digits in doctest results!

Note that in the output of `t` and `u` we write `u` with 14 digits. Writing all 16 digits is not a good idea: if the tests are run on different hardware, round-off errors might be different, and the `doctest` module detects that the numbers are not precisely the same and reports failures. In the present application, where $0 < u(t) \leq 0.8$, we expect round-off errors to be of size 10^{-16}, so comparing 15 digits would probably be reliable, but we compare 14 to be on the safe side. On the other hand, comparing a small number of digits may hide software errors.

Doctests are highly encouraged as they do two things: 1) demonstrate how a function is used and 2) test that the function works.

Caution.

Doctests requires careful coding if they use command-line input or print results to the terminal window. Command-line input must be simulated by filling `sys.argv` correctly, e.g., `sys.argv = '--I 1.0 --a 5'.split`. The output lines of print statements inside doctests must be copied exactly as they appear when running the statements in an interactive Python shell.

3.2 Unit tests and test functions

The unit testing technique consists of identifying small units of code, say a function, and write one or more tests for each unit. One test should, ideally, not depend on the outcome of other tests. The recommended practice is actually to design and write the unit tests first and *then* implement the functions!

In scientific computing it is not always obvious how to best perform unit testing. The units is naturally larger than in non-scientific software. Very often the solution procedure of a mathematical problem identifies a unit, such as our `solver` function.

Two Python test frameworks: nose and pytest. Python offers two very easy-to-use software frameworks for implementing unit tests: nose and pytest. These work (almost) in the same way, but my recommendation is to go for pytest.

Test function requirements. Each test can in these frameworks be realized as a *test function* that follows three rules:

1. The name must start with `test_`.

2. Function arguments are not allowed.

3. An `AssertionError` exception must be raised if the test fails.

A specific example might be illustrative before proceeding. We have the following function that we want to test:

```
def double(n):
    return 2*n
```

The corresponding test function may look like this:

```
def test_double():
    """Test that double(n) works for one specific n."""
    n = 4
    expected = 2*4
    computed = double(4)
    if expected != computed:
        raise AssertionError
```

The last two lines, however, are never written like this in test functions. Instead, use Python's `assert` statement: `assert success, msg`, where `success` is a boolean variable, which is `False` if the test fails, and `msg` is *an optional* message string that is printed when the test fails. A better version of the test function is therefore

```
def test_double():
    """Test that double(n) works for one specific n."""
    n = 4
    expected = 2*4
    computed = double(4)
    msg = 'expected %g, computed %g' % (expected, computed)
    success = expected == computed
    assert success, msg
```

Comparison of real numbers. In scientific computing we very often have to deal with real numbers and finite precision arithmetics (round-off errors) so the `==` operator is not suitable for checking that a test passes. Instead, we must be replace `==` by a comparison based on a tolerance. Here is a slightly different function that we want to test:

```
def third(x):
    return x/3.
```

We write a test function where the expected result is computed as $\frac{1}{3}x$ rather than $x/3$:

```
def test_third():
    """Check that third(x) works for many x values."""
    for x in np.linspace(0, 1, 21):
        expected = (1/3.)*x
        computed = third(x)
        success = expected == computed
        assert success
```

This `test_third` function executes silently, i.e., no failure, until `x` becomes 0.15. Then round-off errors make the `==` comparison `False`. In fact, seven `x` values face this problem. The solution is to compare `expected` and `computed` with a small tolerance:

```
def test_third():
    """Check that third(x) works for many x values."""
    for x in np.linspace(0, 1, 21):
        expected = (1/3.)*x
        computed = third(x)
        tol = 1E-15
        success = abs(expected - computed) < tol
        assert success
```

Real numbers should never be compared with the `==` operator, but always with the absolute value of the difference and a tolerance.

Special assert functions from nose. The nose test framework contains more tailored *assert functions* that can be called instead of using the `assert` statement. For example, comparing two objects within a tolerance, as in the present case, can be done by `assert_almost_equal`:

```
import nose.tools as nt

def test_third():
    x = 0.15
    expected = (1/3.)*x
    computed = third(x)
    nt.assert_almost_equal(
        expected, computed, delta=1E-15,
        msg='diff=%.17E' % (expected - computed))
```

Whether to use the plain `assert` statement with a comparison based on a tolerance or to use the ready-made function `assert_almost_equal` depends on the programmer's preference. The pytest framework (which we recommend) sticks to the plain `assert` statement.

Locating test functions. Test functions can reside in a module together with the functions they are supposed to verify, or the test functions can be collected in separate files having names starting with `test`. Actually, nose and pytest can automatically recursively run all test functions in all `test*.py` files in the current and all subdirectories!

The `decay.py` module file features test functions in the module, but we could equally well have made a subdirectory `tests` and put the test functions in `tests/test_decay.py`.

Running tests. To run all test functions in the file `decay.py` do

```
Terminal> nosetests -s -v decay.py
Terminal> py.test -s -v decay.py
```

The `-s` option ensures that output from the test functions is printed in the terminal window, and `-v` prints the outcome of each individual test function.

Alternatively, if the test functions are in some separate `test*.py` files, we can just write

```
Terminal> py.test -s -v
```

to *recursively* run *all* test functions in the current directory tree. The corresponding

```
Terminal> nosetests -s -v
```

command does the same, but requires subdirectory names to start with `test` or end with `_test` or `_tests` (which is a good habit anyway). An example of a `tests` directory with a `test*.py` file is found in src/softeng1/tests.

Installing nose and pytest. With `pip` available, it is trivial to install nose and/or pytest: `sudo pip install nose` and `sudo pip install pytest`.

3.3 Test function for the solver

Finding good test problems for verifying the implementation of numerical methods is a topic on its own. The challenge is that we very seldom know what the numerical errors are. For the present model problem (1)-(2) solved by (3) one can, fortunately, derive a formula for the numerical approximation:

$$u^n = I\left(\frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\right)^n .$$

Then we know that the implementation should produce numbers that agree with this formula to machine precision. The formula for u^n is known as an *exact discrete solution* of the problem and can be coded as

```
def exact_discrete_solution(n, I, a, theta, dt):
    """Return exact discrete solution of the numerical schemes."""
    dt = float(dt)  # avoid integer division
    A = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*A**n
```

A test function can evaluate this solution on a time mesh and check that the `u` values produced by the `solver` function do not deviate with more than a small tolerance:

```
def test_exact_discrete_solution():
    """Check that solver reproduces the exact discr. sol."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    Nt = int(8/dt)  # no of steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)

    # Evaluate exact discrete solution on the mesh
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(Nt+1)])

    # Find largest deviation
    diff = np.abs(u_de - u).max()
    tol = 1E-14
    success = diff < tol
    assert success
```

An important topic to address in test functions is potentially problematic input to functions. For example, if a, Δt, and θ are integers, one may face problems with unintended integer division in the numerical solution algorithm for the present mathematical problem. We should therefore add a test to make sure our `solver` function does not fall into this potential trap:

```
def test_potential_integer_division():
    """Choose variables that can trigger integer division."""
    theta = 1; a = 1; I = 1; dt = 2
    Nt = 4
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(Nt+1)])
    diff = np.abs(u_de - u).max()
    assert diff < 1E-14
```

3.4 Test function for reading positional command-line arguments

The function `read_command_line_positional` extracts numbers from the command line. To test it, decide on a set of numbers, fill `sys.argv` appropriately, and check that we get the expected numbers:

```
def test_read_command_line_positional():
    # Decide on a data set of input parameters
    I = 1.6;  a = 1.8;  T = 2.2;  theta = 0.5
    dt_values = [0.1, 0.2, 0.05]
    # Expected return from read_command_line_positional
    expected = [I, a, T, theta, dt_values]
    # Construct corresponding sys.argv array
    sys.argv = [sys.argv[0], str(I), str(a), str(T), 'CN'] + \
               [str(dt) for dt in dt_values]
    computed = read_command_line_positional()
    for expected_arg, computed_arg in zip(expected, computed):
        assert expected_arg == computed_arg
```

Note that `sys.argv[0]` is always the program name and that we have to copy that string from the original `sys.argv` array to the new one we construct in the

test function. (Actually, the test function destroys the original `sys.argv` that Python fetched from the command line.)

Any numerical code writer should always be skeptical to the use of the exact equality operator `==` in test functions, since round-off errors often come into play. Here, however, we set some real values, convert them to strings and convert back again to real numbers (of the same precision). This string-number conversion does not involve any finite precision arithmetics effects so we can safely use `==` in tests. Note also that the last element in `expected` and `computed` is the list `dt_values`, and `==` works for comparing two lists too.

3.5 Test function for reading option-value pairs

Testing the function `read_command_line_argparse` follows the set up for the similar function for positional command-line arguments. However, the construction of the command line is a bit more complicated. We find it convenient to construct the line as a string and then split the line into words to get the desired list `sys.argv`:

```
def test_read_command_line_argparse():
    I = 1.6;  a = 1.8;  T = 2.2;  theta = 0.5
    dt_values = [0.1, 0.2, 0.05]
    # Expected return from read_command_line_positional
    expected = [I, a, T, theta, dt_values]
    # Construct corresponding sys.argv array
    cml = '%s --a %s --I %s --T %s --scheme CN --dt ' % \
          (sys.argv[0], a, I, T)
    cml = cml + ' '.join([str(dt) for dt in dt_values])
    sys.argv = cml.split()
    computed = read_command_line_argparse()
    for expected_arg, computed_arg in zip(expected, computed):
        assert expected_arg == computed_arg
```

Let silent test functions speak up during development!

When you develop test functions in a module, it is common to use IPython to reload the module and call the test function as it gets developed:

```
In[1]: import decay

In[2]: decay.test_read_command_line_argparse()

In[3]: reload(decay)  # force new import

In[2]: decay.test_read_command_line_argparse()  # test again
```

However, a working test function is completely silent! Many find it psychologically annoying to convince themselves that a completely silent function is doing the right things. It can therefore, during development of a test function, be convenient to insert print statements in the function to monitor

that the function body is indeed executed. For example, one can print the expected and computed values in the terminal window.

3.6 Classical class-based unit testing

The test functions written for the nose and pytest frameworks are very straightforward and to the point, with no framework-required boilerplate code. We just write the statements we need to make the computations and comparisons and then apply the required `assert`.

The classical way of implementing unit tests (which derives from the JUnit object-oriented tool in Java) leads to much more comprehensive implementations with much more boilerplate code. Python comes with a built-in module `unittest` for doing this type of classical unit tests. Although I strongly recommend to use nose or pytest over `unittest`, class-based unit testing in the style of `unittest` has a very strong position in computer science and is so widespread that even computational scientists should have an idea how such unit test code is written. A short demo of `unittest` is therefore included next.

Suppose we have a function `double(x)` in a module file `mymod.py`:

```
def double(x):
    return 2*x
```

Unit testing with the aid of the `unittest` module consists of writing a file `test_mymod.py` for testing the functions in `mymod.py`. The individual tests must be methods with names starting with `test_` in a class derived from class `TestCase` in `unittest`. With one test method for the function `double`, the `test_mymod.py` file becomes

```
import unittest
import mymod

class TestMyCode(unittest.TestCase):
    def test_double(self):
        x = 4
        expected = 2*x
        computed = mymod.double(x)
        self.assertEqual(expected, computed)

if __name__ == '__main__':
    unittest.main()
```

The test is run by executing the test file `test_mymod.py` as a standard Python program. There is no support in `unittest` for automatically locating and running all tests in all test files in a directory tree.

We could use the basic `assert` statement as we did with nose and pytest functions, but those who write code based on `unittest` almost exclusively use the wide range of built-in assert functions such as `assertEqual`, `assertNotEqual`, `assertAlmostEqual`, to mention some of them.

Translation of `test_exact_discrete_solution`, `test_potential_integer_division`, and the other test functions in `decay.py` to `unittest` means making a new file `test_decay.py` file with a test class `TestDecay` where the stand-alone functions for nose/pytest now become methods in this class.

```
import unittest
import decay
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    ...

class TestDecay(unittest.TestCase):

    def test_exact_discrete_solution(self):
        theta = 0.8; a = 2; I = 0.1; dt = 0.8
        Nt = int(8/dt)  # no of steps
        u, t = decay.solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
        # Evaluate exact discrete solution on the mesh
        u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                         for n in range(Nt+1)])
        diff = np.abs(u_de - u).max()  # largest deviation
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_potential_integer_division(self):
        ...
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_read_command_line_positional(self):
        ...
        for expected_arg, computed_arg in zip(expected, computed):
            self.assertEqual(expected_arg, computed_arg)

    def test_read_command_line_argparse(self):
        ...

if __name__ == '__main__':
    unittest.main()
```

4 Classes for problem and solution method

The numerical solution procedure was compactly and conveniently implemented in a Python function `solver` in Section 1.1. In more complicated problems it might be beneficial to use classes instead of functions only. Here we shall describe a class-based software design well suited for scientific problems where there is a mathematical model of some physical phenomenon and some numerical methods to solve the equations involved in the model.

We introduce a class `Problem` to hold the definition of the physical problem, and a class `Solver` to hold the data and methods needed to numerically solve the problem. The forthcoming text will explain the inner workings of these classes and how they represent an alternative to the `solver` and `experiment_*` functions in the `decay` module.

Explaining the details of class programming in Python is considered far beyond the scope of this text. Readers who are unfamiliar with Python class programming should first consult one of the many electronic Python tutorials or textbooks to come up to speed with concepts and syntax of Python classes before reading on. The author has a gentle introduction to class programming for scientific applications in [1], see Chapter 7 and 9 and Appendix E. Other useful resources are

- The Python Tutorial: `http://docs.python.org/2/tutorial/classes.html`
- Wiki book on Python Programming: `http://en.wikibooks.org/wiki/Python_Programming/Classes`
- `tutorialspoint.com`: `http://www.tutorialspoint.com/python/python_classes_objects.htm`

4.1 The problem class

The purpose of the problem class is to store all information about the mathematical model. This usually means the physical parameters and formulas in the problem. Looking at our model problem (1)-(2), the physical data cover I, a, and T. Since we have an analytical solution of the ODE problem, we may add this solution in terms of a Python function (or method) to the problem class as well. A possible problem class is therefore

```
from numpy import exp

class Problem(object):
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def u_exact(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

We could in the `u_exact` method have written `self.I*exp(-self.a*t)`, but using local variables `I` and `a` allows the nicer formula `I*exp(-a*t)`, which looks much closer to the mathematical expression Ie^{-at}. This is not an important issue with the current compact formula, but is beneficial in more complicated problems with longer formulas to obtain the closest possible relationship between code and mathematics. My coding style is to strip off the `self` prefix when the code expresses mathematical formulas.

The class data can be set either as arguments in the constructor or at any time later, e.g.,

```
problem = Problem(T=5)
problem.T = 8
problem.dt = 1.5
```

(Some programmers prefer `set` and `get` functions for setting and getting data in classes, often implemented via *properties* in Python, but I consider that overkill when we just have a few data items in a class.)

It would be convenient if class `Problem` could also initialize the data from the command line. To this end, we add a method for defining a set of command-line options and a method that sets the local attributes equal to what was found on the command line. The default values associated with the command-line options are taken as the values provided to the constructor. Class `Problem` now becomes

```
class Problem(object):
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def define_command_line_options(self, parser=None):
        """Return updated (parser) or new ArgumentParser object."""
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        parser.add_argument(
            '--I', '--initial_condition', type=float,
            default=1.0, help='initial condition, u(0)',
            metavar='I')
        parser.add_argument(
            '--a', type=float, default=1.0,
            help='coefficient in ODE', metavar='a')
        parser.add_argument(
            '--T', '--stop_time', type=float,
            default=1.0, help='end time of simulation',
            metavar='T')
        return parser

    def init_from_command_line(self, args):
        """Load attributes from ArgumentParser into instance."""
        self.I, self.a, self.T = args.I, args.a, args.T

    def u_exact(self, t):
        """Return the exact solution u(t)=I*exp(-a*t)."""
        I, a = self.I, self.a
        return I*exp(-a*t)
```

Observe that if the user already has an `ArgumentParser` object it can be supplied, but if she does not have any, class `Problem` makes one. Python's `None` object is used to indicate that a variable is not initialized with a proper value.

4.2 The solver class

The solver class stores parameters related to the numerical solution method and provides a function `solve` for solving the problem. For convenience, a problem object is given to the constructor such a solver object such that the latter also get access to physical data. In the present example, the numerical solution method involves the parameters Δt and θ, which then constitute the data part of the solver class. We include, as in the problem class, functionality for reading Δt and θ from the command line:

```
class Solver(object):
    def __init__(self, problem, dt=0.1, theta=0.5):
        self.problem = problem
        self.dt, self.theta = float(dt), theta

    def define_command_line_options(self, parser):
        """Return updated (parser) or new ArgumentParser object."""
        parser.add_argument(
            '--scheme', type=str, default='CN',
            help='FE, BE, or CN')
        parser.add_argument(
            '--dt', '--time_step_values', type=float,
            default=[1.0], help='time step values',
            metavar='dt', nargs='+', dest='dt_values')
        return parser

    def init_from_command_line(self, args):
        """Load attributes from ArgumentParser into instance."""
        self.dt, self.theta = args.dt, args.theta

    def solve(self):
        self.u, self.t = solver(
            self.problem.I, self.problem.a, self.problem.T,
            self.dt, self.theta)

    def error(self):
        """Return norm of error at the mesh points."""
        u_e = self.problem.u_exact(self.t)
        e = u_e - self.u
        E = np.sqrt(self.dt*np.sum(e**2))
        return E
```

Note that we see no need to repeat the body of the previously developed and tested `solver` function. We just call that function from the `solve` method. In this way, class `Solver` is merely a class wrapper of the stand-alone `solver` function. With a single object of class `Solver` we have all the physical and numerical data bundled together with the numerical solution method.

Combining the objects. Eventually we need to show how the classes `Problem` and `Solver` play together. We read parameters from the command line and make a plot with the numerical and exact solution:

```
def experiment_classes():
    problem = Problem()
    solver = Solver(problem)

    # Read input from the command line
    parser = problem.define_command_line_options()
    parser = solver. define_command_line_options(parser)
    args = parser.parse_args()
    problem.init_from_command_line(args)
    solver. init_from_command_line(args)

    # Solve and plot
    solver.solve()
    import matplotlib.pyplot as plt
    t_e = np.linspace(0, T, 1001)    # very fine mesh for u_e
```

```
    u_e = problem.u_exact(t_e)
    print 'Error:', solver.error()

    plt.plot(t,   u,   'r--o')       # dashed red line with circles
    plt.plot(t_e, u_e, 'b-')         # blue line for u_e
    plt.legend(['numerical, theta=%g' % theta, 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf')
    plt.show()
```

4.3 Improving the problem and solver classes

The previous `Problem` and `Solver` classes containing parameters soon get much repetitive code when the number of parameters increases. Much of this code can be parameterized and be made more compact. For this purpose, we decide to collect all parameters in a dictionary, `self.prm`, with two associated dictionaries `self.type` and `self.help` for holding associated object types and help strings. The reason is that processing dictionaries is easier than processing a set of individual attributes. For the specific ODE example we deal with, the three dictionaries in the problem class are typically

```
self.prm  = dict(I=1, a=1, T=10)
self.type = dict(I=float, a=float, T=float)
self.help = dict(I='initial condition, u(0)',
                 a='coefficient in ODE',
                 T='end time of simulation')
```

Provided a problem or solver class defines these three dictionaries in the constructor, we can create a super class `Parameters` with general code for defining command-line options and reading them as well as methods for setting and getting each parameter. A `Problem` or `Solver` for a particular mathematical problem can then inherit most of the needed functionality and code from the `Parameters` class. For example,

```
class Problem(Parameters):
    def __init__(self):
        self.prm  = dict(I=1, a=1, T=10)
        self.type = dict(I=float, a=float, T=float)
        self.help = dict(I='initial condition, u(0)',
                         a='coefficient in ODE',
                         T='end time of simulation')

    def u_exact(self, t):
        I, a = self['I a'.split()]
        return I*np.exp(-a*t)

class Solver(Parameters):
    def __init__(self, problem):
        self.problem = problem   # class Problem object
        self.prm  = dict(dt=0.5, theta=0.5)
        self.type = dict(dt=float, theta=float)
```

```
        self.help = dict(dt='time step value',
                         theta='time discretization parameter')

    def solve(self):
        from decay import solver
        I, a, T = self.problem['I a T'.split()]
        dt, theta = self['dt theta'.split()]
        self.u, self.t = solver(I, a, T, dt, theta)
```

By inheritance, these classes can automatically do a lot more when it comes to reading and assigning parameter values:

```
problem = Problem()
solver = Solver(problem)

# Read input from the command line
parser = problem.define_command_line_options()
parser = solver. define_command_line_options(parser)
args = parser.parse_args()
problem.init_from_command_line(args)
solver. init_from_command_line(args)

# Other syntax for setting/getting parameter values
problem['T'] = 6
print 'Time step:', solver['dt']

solver.solve()
u, t = solver.u, solver.t
```

A generic class for parameters. A simplified version of the parameter class looks as follows:

```
class Parameters(object):
    def __getitem__(self, name):
        """obj[name] syntax for getting parameters."""
        if isinstance(name, (list,tuple)):         # get many?
            return [self.prm[n] for n in name]
        else:
            return self.prm[name]

    def __setitem__(self, name, value):
        """obj[name] = value syntax for setting a parameter."""
        self.prm[name] = value

    def define_command_line_options(self, parser=None):
        """Automatic registering of options."""
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        for name in self.prm:
            tp = self.type[name] if name in self.type else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                '--' + name, default=self.get(name), metavar=name,
                type=tp, help=help)
```

```
        return parser

    def init_from_command_line(self, args):
        for name in self.prm:
            self.prm[name] = getattr(args, name)
```

The file `decay_oo.py` contains a slightly more advanced version of class `Parameters` where we in the functions for getting and setting parameters test for valid parameter names and raise exceptions with informative messages if any name is not registered.

We have already sketched the `Problem` and `Solver` classes that build on inheritance from `Parameters`. We have also shown how they are used. The only remaining code is to make the plot, but this code is identical to previous versions when the numerical solution is available in an object `t` and the exact one in `u_e`.

The advantage with the `Parameters` class is that it scales to problems with a large number of physical and numerical parameters: as long as the parameters are defined once via a dictionary, the compact code in class `Parameters` can handle any collection of parameters of any size.

5 Automating scientific experiments

Empirical scientific investigations based on running computer programs require careful design of the experiments and accurate reporting of results. Although there is a strong tradition to do such investigations manually, the extreme requirements to scientific accuracy make a program much better suited to conduct the experiments. We shall in this section outline how we can write such programs, often called *scripts*, for running other programs and archiving the results.

Scientific investigation.

The purpose of the investigations is to explore the quality of numerical solutions to an ordinary differential equation. More specifically, we solve the initial-value problem

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T], \tag{4}$$

by the θ-rule:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t} u^n, \quad u^0 = I\,. \tag{5}$$

This scheme corresponds to well-known methods: $\theta = 0$ gives the Forward Euler (FE) scheme, $\theta = 1$ gives the Backward Euler (BE) scheme, and $\theta = \frac{1}{2}$ gives the Crank-Nicolson (CN) or midpoint/centered scheme.

For chosen constants I, a, and T, we run the three schemes for various values of Δt, and present in a report the following results:

1. visual comparison of the numerical and exact solution in a plot for each Δt and $\theta = 0, 1, \frac{1}{2}$,
2. a table and a plot of the norm of the numerical error versus Δt for $\theta = 0, 1, \frac{1}{2}$.

The report will also document the mathematical details of the problem under investigation.

5.1 Available software

Appropriate software for implementing (5) is available in a program `model.py`, which is run as

```
Terminal> python model.py --I 1.5 --a 0.25 --T 6 --dt 1.25 0.75 0.5
```

The command-line input corresponds to setting $I = 1.5$, $a = 0.25$, $T = 6$, and run three values of Δt: 1.25, 0.75, ad 0.5.

The results of running this `model.py` command are text in the terminal window and a set of plot files. The plot files have names `M_D.E`, where `M` denotes the method (`FE`, `BE`, `CN` for $\theta = 0, 1, \frac{1}{2}$, respectively), `D` the time step length (here `1.25`, `0.75`, or `0.5`), and `E` is the plot file extension `png` or `pdf`. The text output in the terminal window looks like

```
0.0   1.25:    5.998E-01
0.0   0.75:    1.926E-01
0.0   0.50:    1.123E-01
0.0   0.10:    1.558E-02
0.5   1.25:    6.231E-02
0.5   0.75:    1.543E-02
0.5   0.50:    7.237E-03
0.5   0.10:    2.469E-04
1.0   1.25:    1.766E-01
1.0   0.75:    8.579E-02
1.0   0.50:    6.884E-02
1.0   0.10:    1.411E-02
```

The first column is the θ value, the next the Δt value, and the final column represents the numerical error E (the norm of discrete error function on the mesh).

5.2 Required new results

The results we need for our investigations are slightly different than what is directly produced by `model.py`:

1. We need to collect all the plots for one numerical method (FE, BE, CN) in a single plot. For example, if 4 Δt values are run, the summarizing plot for the BE method has 2×2 subplots, with the subplot corresponding to the largest Δt in the upper left corner and the smallest in the bottom right corner.
2. We need to create a table containing Δt values in the first column and the numerical error E for $\theta = 0, 0.5, 1$ in the next three columns. This table should be available as a standard CSV file.
3. We need to plot the numerical error E versus Δt in a log-log plot.

Consequently, we must write a script that can run `model.py` as described and produce the results 1-3 above. This requires combining multiple plot files into one file and interpreting the output from `model.py` as data for plotting and file storage.

If the script's name is `exper1.py`, we run it with the desired Δt values as positional command-line arguments:

```
Terminal> python exper1.py 0.5 0.25 0.1 0.05
```

This run will then generate eight plot files: `FE.png` and `FE.pdf` summarizing the plots with the FE method, `BE.png` and `BE.pdf` with the BE method, `CN.png` and `CN.pdf` with the CN method, and `error.png` and `error.pdf` with the log-log plot of the numerical error versus Δt. In addition, the table with numerical errors is written to a file `error.csv`.

Reproducible and replicable science.

A script that automates running our computer experiments will ensure that the experiments can easily be rerun by anyone in the future, either to confirm the same results or redo the experiments with other input data. Also, whatever we did to produce the results is documented in every detail in the script.

A project where anyone can easily repeat the experiments with the same data is referred to as being *replicable*, and replicability should be a fundamental requirement in scientific computing work. Of more scientific interest is *reproducibilty*, which means that we can also run alternative experiments to arrive at the same conclusions. This requires more than an automating script.

5.3 Combining plot files

The script for running experiments needs to combine multiple image files into one. The `montage` and `convert` programs in the ImageMagick software suite

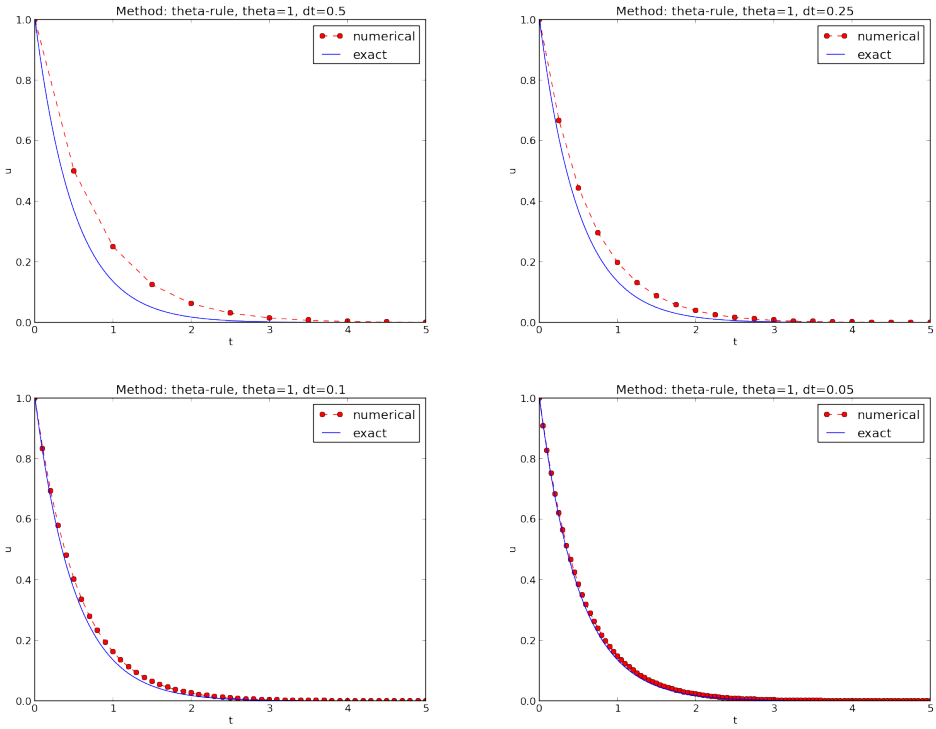


Figure 3: Illustration of the Backward Euler method for four time step values.

can be used to combine image files. However, these programs are best suited for PNG files. For vector plots in PDF format one needs other tools to preserve the quality: `pdftk`, `pdfnup`, and `pdfcrop`.

Suppose you have four files `f1.png`, `f2.png`, `f3.png`, and `f4.png` and want to combine them into a 2×2 table of subplots in a new file `f.png`, see Figure 3 for an example.

The appropriate ImageMagick commands are

```
Terminal> montage -background white -geometry 100% -tile 2x \
          f1.png f2.png f3.png f4.png f.png
Terminal> convert -trim f.png f.png
Terminal> convert f.png -transparent white f.png
```

The first command mounts the four files in one, the next `convert` command removes unnecessary surrounding white space, and the final `convert` command makes the white background transparent.

High-quality montage of PDF files `f1.pdf`, `f2.pdf`, `f3.pdf`, and `f4.pdf` into `f.pdf` goes like

```
Terminal> pdftk f1.pdf f2.pdf f3.pdf f4.pdf output tmp.pdf
Terminal> pdfnup --nup 2x2 --outfile tmp.pdf tmp.pdf
Terminal> pdfcrop tmp.pdf f.pdf
Terminal> rm -f tmp.pdf
```

5.4 Running a program from Python

The script for automating experiments needs to run the `model.py` program with appropriate command-line options. Python has several tools for executing an arbitrary command in the operating systems. Let `cmd` be a string containing the desired command. In the present case study, `cmd` could be `'python model.py --I 1 --dt 0.5 0.2'`. The following code executes `cmd` and loads the text output into a string `output`:

```
from subprocess import Popen, PIPE, STDOUT
p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
output, dummy = p.communicate()

# Check if the execution was successful
failure = p.returncode
if failure:
    print 'Command failed:', cmd; sys.exit(1)
```

Unsuccessful execution usually makes it meaningless to continue the program, and therefore we abort the program with `sys.exit(1)`. Any argument different from 0 signifies to the computer's operating system that our program stopped with a failure.

We need to interpret the contents of the string `output` and store the data in an appropriate data structure for further processing. Since the content is basically a table and will be transformed to a spread sheet format, we let the columns in the table be represented by lists in the program, and we collect these columns in a dictionary whose keys are natural column names: `dt` and the three values of θ. The following code translates the output of `cmd` (`output`) to such a dictionary of lists (`errors`):

```
errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
for line in output.splitlines():
    words = line.split()
    if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
        # typical line: 0.0   1.25:    7.463E+00
        theta = float(words[0])
        E = float(words[2])
        errors[theta].append(E)
```

5.5 The automating script

We have now all the core elements in place to write the complete script where we run `model.py` for a set of Δt values (given as positional command-line

arguments), make the error plot, write the CSV file, and combine plot files as described above. The complete code is listed below, followed by some explaining comments.

```
import os, sys, glob
import matplotlib.pyplot as plt

def run_experiments(I=1, a=2, T=5):
    # The command line must contain dt values
    if len(sys.argv) > 1:
        dt_values = [float(arg) for arg in sys.argv[1:]]
    else:
        print 'Usage: %s dt1 dt2 dt3 ...' %  sys.argv[0]
        sys.exit(1)  # abort

    # Run module file and grab output
    cmd = 'python model.py --I %g --a %g --T %g' % (I, a, T)
    dt_values_str = ' '.join([str(v) for v in dt_values])
    cmd += ' --dt %s' % dt_values_str
    print cmd
    from subprocess import Popen, PIPE, STDOUT
    p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
    output, dummy = p.communicate()
    failure = p.returncode
    if failure:
        print 'Command failed:', cmd; sys.exit(1)

    errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
    for line in output.splitlines():
        words = line.split()
        if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
            # typical line: 0.0   1.25:    7.463E+00
            theta = float(words[0])
            E = float(words[2])
            errors[theta].append(E)

    # Find min/max for the axis
    E_min = 1E+20; E_max = -E_min
    for theta in 0, 0.5, 1:
        E_min = min(E_min, min(errors[theta]))
        E_max = max(E_max, max(errors[theta]))

    plt.loglog(errors['dt'], errors[0], 'ro-')
    plt.loglog(errors['dt'], errors[0.5], 'b+-')
    plt.loglog(errors['dt'], errors[1], 'gx-')
    plt.legend(['FE', 'CN', 'BE'], loc='upper left')
    plt.xlabel('log(time step)')
    plt.ylabel('log(error)')
    plt.axis([min(dt_values), max(dt_values), E_min, E_max])
    plt.title('Error vs time step')
    plt.savefig('error.png');  plt.savefig('error.pdf')

    # Write out a table in CSV format
    f = open('error.csv', 'w')
    f.write(r'$\Delta t$,$\theta=0$,$\theta=0.5$,$\theta=1$' + '\n')
    for _dt, _fe, _cn, _be in zip(
        errors['dt'], errors[0], errors[0.5], errors[1]):
        f.write('%.2f,%.4f,%.4f,%.4f\n' % (_dt, _fe, _cn, _be))
    f.close()
```

```
    # Combine images into rows with 2 plots in each row
    image_commands = []
    for method in 'BE', 'CN', 'FE':
        pdf_files = ' '.join(['%s_%g.pdf' % (method, dt)
                              for dt in dt_values])
        png_files = ' '.join(['%s_%g.png' % (method, dt)
                              for dt in dt_values])
        image_commands.append(
            'montage -background white -geometry 100%' +
            ' -tile 2x %s %s.png' % (png_files, method))
        image_commands.append(
            'convert -trim %s.png %s.png' % (method, method))
        image_commands.append(
            'convert %s.png -transparent white %s.png' %
            (method, method))
        image_commands.append(
            'pdftk %s output tmp.pdf' % pdf_files)
        num_rows = int(round(len(dt_values)/2.0))
        image_commands.append(
            'pdfnup --nup 2x%d --outfile tmp.pdf tmp.pdf' % num_rows)
        image_commands.append(
            'pdfcrop tmp.pdf %s.pdf' % method)

    for cmd in image_commands:
        print cmd
        failure = os.system(cmd)
        if failure:
            print 'Command failed:', cmd; sys.exit(1)

    # Remove the files generated above and by model.py
    from glob import glob
    filenames = glob('*_*.png') + glob('*_*.pdf') + glob('tmp*.pdf')
    for filename in filenames:
        os.remove(filename)

if __name__ == '__main__':
    run_experiments(I=1, a=2, T=5)
    plt.show()
```

We may comment upon many useful constructs in this script:

- `[float(arg) for arg in sys.argv[1:]]` builds a list of real numbers from all the command-line arguments.
- `['%s_%s.png' % (method, dt) for dt in dt_values]` builds a list of filenames from a list of numbers (`dt_values`).
- All `montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop` commands for creating composite figures are stored in a list and later executed in a loop.
- `glob('*_*.png')` returns a list of the names of all files in the current directory where the filename matches the Unix wildcard notation `*_*.png` (meaning any text, underscore, any text, and then `.png`).
- `os.remove(filename)` removes the file with name `filename`.

- `failure = os.system(cmd)` runs an operating system command with simpler syntax than what is required by `subprocess` (but the output of `cmd` cannot be captured).

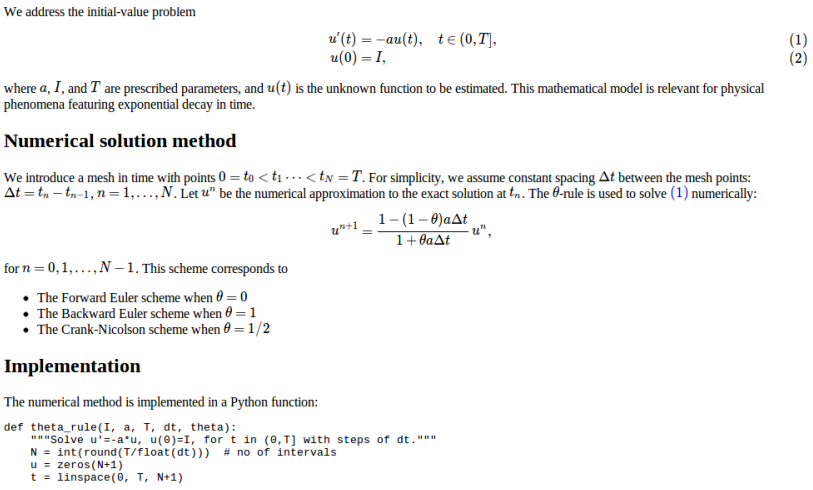
5.6 Making a report

The results of running computer experiments are best documented in a little report containing the problem to be solved, key code segments, and the plots from a series of experiments. At least the part of the report containing the plots should be automatically generated by the script that performs the set of experiments, because in that script we know exactly which input data that were used to generate a specific plot, thereby ensuring that each figure is connected to the right data. Take a look at a sample report to see what we have in mind.

Word, OpenOffice, GoogleDocs. Microsoft Word, its open source counterparts OpenOffice and LibreOffice, along with GoogleDocs and similar online services are the dominating tools for writing reports today. Nevertheless, scientific reports often need mathematical equations and nicely typeset computer code in monospace font. The support for mathematics and computer code in the mentioned tools is in this author's view not on par with the technologies based on *markup languages* and which are addressed below. Also, with markup languages one has a readable, pure text file as source for the report, and changes in this text can easily be tracked by version control systems like Git. The result is a very strong tool for monitoring "who did what when" with the files, resulting in increased reliability of the writing process. For collaborative writing, the merge functionality in Git leads to safer simultaneously editing that what is offered even by collaborative tools like GoogleDocs.

HTML with MathJax. HTML is the markup language used for web pages. Nicely typeset computer code is straightforward in HTML, and high-quality mathematical typesetting is available using an extension to HTML called MathJax, which allows formulas and equations to be typeset with LaTeX syntax and nicely rendered in web browsers, see Figure 4. A relatively small subset of LaTeX environments for mathematics is supported, but the syntax for formulas is quite rich. Inline formulas look like `\( u'=-au \)` while equations are surrounded by `$$` signs. Inside such signs, one can use `\[ u'=-au \]` for unnumbered equations, or `\begin{equation}` and `\end{equation}` for numbered equations, or `\begin{align}` and `\end{align}` for multiple numbered aligned equations. You need to be familiar with mathematical typesetting in LaTeX to write MathJax code.

The file `exper1_mathjax.py` calls a script `exper1.py` to perform the numerical experiments and then runs Python statements for creating an HTML file with the source code for the scientific report.



We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0,T], \tag{1}$$
$$u(0) = I, \tag{2}$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time.

Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N$. Let u^n be the numerical approximation to the exact solution at t_n. The θ-rule is used to solve (1) numerically:

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

- The Forward Euler scheme when $\theta = 0$
- The Backward Euler scheme when $\theta = 1$
- The Crank-Nicolson scheme when $\theta = 1/2$

Implementation

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)
```

Figure 4: Report in HTML format with MathJax.

LaTeX. The *de facto* language for mathematical typesetting and scientific report writing is LaTeX. A number of very sophisticated packages have been added to the language over a period of three decades, allowing very fine-tuned layout and typesetting. For output in the PDF format, see Figure 5 for an example, LaTeX is the definite choice when it comes to *typesetting quality*. The LaTeX language used to write the reports has typically a lot of commands involving backslashes and braces, and many claim that LaTeX syntax is not particularly readable. For output on the web via HTML code (i.e., not only showing the PDF in the browser window), LaTeX struggles with delivering high quality typesetting. Other tools, especially Sphinx, give better results and can also produce nice-looking PDFs. The file `exper1_latex.py` shows how to generate the LaTeX source from a program.

Sphinx. Sphinx is a typesetting language with similarities to HTML and LaTeX, but with much less tagging. It has recently become very popular for software documentation and mathematical reports. Sphinx can utilize LaTeX for mathematical formulas and equations. Unfortunately, the subset of LaTeX mathematics supported is less than in full MathJax (in particular, numbering of multiple equations in an `align` type environment is not supported). The Sphinx syntax is an extension of the reStructuredText language. An attractive feature of Sphinx is its rich support for fancy layout of web pages. In particular, Sphinx can easily be combined with various layout *themes* that give a certain look and feel to the web site and that offers table of contents, navigation, and search facilities, see Figure 6.

3 Implementation

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    for n in range(0, N):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

4 Numerical experiments

We define a set of numerical experiments where I, a, and T are fixed, while Δt and θ are varied. In particular, $I = 1$, $a = 2$, $\Delta t = 1.25, 0.75, 0.5, 0.1$.

Figure 5: Report in PDF format generated from LaTeX source.

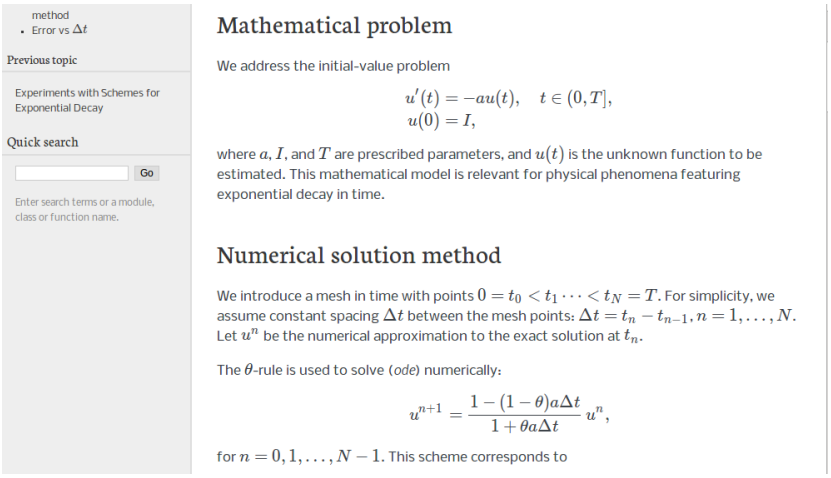


Figure 6: Report in HTML format generated from Sphinx source.

Markdown. A recent, very popular format for easy writing of web pages is Markdown. Text is written very much like one would do in email, using spacing and special characters to naturally format the code instead of heavily tagging the text as in LaTeX and HTML. With the tool Pandoc one can go from Markdown to a variety of formats. HTML is a common output format, but LaTeX, epub, XML, OpenOffice/LibreOffice, MediaWiki, and Microsoft Word are some other possibilities. A Markdown version of our scientific report demo is available as an IPython/Jupyter notebook (described next).

IPython/Jupyter notebooks. The IPython Notebook is a web-based tool where one can write scientific reports with live computer code and graphics. Or the other way around: software can be equipped with documentation in the style of scientific reports. A slightly extended version of Markdown is used for writing text and mathematics, and the source code of a notebook is in json format. The interest in the notebook has grown amazingly fast over just a few years, and further development now takes place in the Jupyter project, which supports a lot of programming languages for interactive notebook computing. Jupyter notebooks are primarily live electronic documents, but they can be printed out as PDF reports too. A notebook version of our scientific report can be downloaded and experimented with or just statically viewed in a browser.

Wiki formats. A range of wiki formats are popular for creating notes on the web, especially documents which allow groups of people to edit and add content. Apart from MediaWiki (the wiki format used for Wikipedia), wiki formats have no support for mathematical typesetting and also limited tools for displaying computer code in nice ways. Wiki formats are therefore less suitable for scientific reports compared to the other formats mentioned here.

DocOnce. Since it is difficult to choose the right tool or format for writing a scientific report, it is advantageous to write the content in a format that easily translates to LaTeX, HTML, Sphinx, Markdown, IPython/Jupyter notebooks, and various wikis. DocOnce is such a tool. It is similar to Pandoc, but offers some special convenient features for writing about mathematics and programming. The tagging is modest, somewhere between LaTeX and Markdown. The program `exper1_do.py` demonstrates how to generate DocOnce code for a scientific report. There is also a corresponding rich demo of the resulting reports that can be made from this DocOnce code.

5.7 Publishing a complete project

To assist the important principle of *replicable* science, a report documenting scientific investigations should be accompanied by all the software and data used for the investigations so that others have a possibility to redo the work and assess the qualify of the results.

One way of documenting a complete project is to make a directory tree with all relevant files. Preferably, the tree is published at some project hosting site like Bitbucket or GitHub so that others can download it as a tarfile, zipfile, or clone the files directly using the Git version control system. For the investigations outlined in Section 5.6, we can create a directory tree with files

```
setup.py
./src:
   model.py
./doc:
   ./src:
      exper1_mathjax.py
      make_report.sh
```

```
    run.sh
./pub:
    report.html
```

The `src` directory holds source code (modules) to be reused in other projects, the `setup.py` script builds and installs such software, the `doc` directory contains the documentation, with `src` for the source of the documentation (usually written in a markup language) and `pub` for published (compiled) documentation. The `run.sh` file is a simple Bash script listing the `python` commands we used to run `exper1_mathjax.py` to generate the experiments and the `report.html` file.

6 Exercises

Problem 1: Make a tool for differentiating curves

Suppose we have a curve specified through a set of discrete coordinates (x_i, y_i), $i = 0, \ldots, n$, where the x_i values are uniformly distributed with spacing Δx: $x_i = \Delta x$. The derivative of this curve, defined as a new curve with points (x_i, d_i), can be computed via finite differences:

$$d_0 = \frac{y_1 - y_0}{\Delta x}, \tag{6}$$

$$d_i = \frac{y_{i+1} - y_{i-1}}{2\Delta x}, \quad i = 1, \ldots, n-1, \tag{7}$$

$$d_n = \frac{y_n - y_{n-1}}{\Delta x}\,. \tag{8}$$

a) Write a function `differentiate(x, y)` for differentiating a curve with coordinates in the arrays `x` and `y`, using the formulas above. The function should return the coordinate arrays of the resulting differentiated curve.

b) Since the formulas for differentiation used here are only approximate, with unknown approximation errors, it is challenging to construct test cases. Here are three approaches, which should be implemented in three separate test functions.

1. Consider a curve with three points and compute d_i, $i = 0, 1, 2$, by hand. Make a test that compares the hand-calculated results with those from the function in a).
2. The formulas for d_i are exact for points on a straight line, as all the d_i values are then the same, equal to the slope of the line. A test can check this property.
3. For point lying on a parabola, the values for d_i, $i = 1, \ldots, n-1$, should equal the derivative of the parabola. Make a test based on this property too.

c) Start with a curve corresponding to $y = \sin(\pi x)$ and $n + 1$ points in $[0, 1]$. Apply `differentiate` four times and plot the resulting curve and the exact $y = \sin \pi x$ for $n = 6, 11, 21, 41$.
Filename: `curvediff.py`.

Problem 2: Make solid software for the Trapezoidal rule

An integral

$$\int_a^b f(x)dx$$

can be numerically approximated by the Trapezoidal rule,

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) + h\sum_{i=1}^{n-1} f(x_i),$$

where x_i is a set of uniformly spaced points in $[a, b]$:

$$h = \frac{b-a}{n}, \quad x_i = a + ih,\ i = 1, \ldots, n-1\,.$$

Somebody has used this rule to compute the integral $\int_0^\pi \sin^2 x dx$:

```
from math import pi, sin
np = 20
h = pi/np
I = 0
for k in range(1, np):
    I += sin(k*h)**2
print I
```

a) The "flat" implementation above suffers from three serious flaws:

1. A general numerical algorithm (the Trapezoidal rule) is implemented in a specialized form where the formula for f is inserted directly into the code for the general integration formula.
2. A general numerical algorithm is not encapsulated as a general function, with appropriate parameters, which can be reused across a wide range of applications.
3. The lazy programmer dropped the first terms in the general formula since $\sin(0) = \sin(\pi) = 0$.
4. The sloppy programmer used `np` (number of points?) as variable for `n` in the formula and a counter `k` instead of `i`. Such small deviations from the mathematical notation are completely unnecessary. The closer the code and the mathematics can get, the easier it is to spot errors in formulas.

Write a function `trapezoidal` that fixes these flaws. Place the function in a module `trapezoidal`.

b) Write a test function `test_trapezoidal`. Call the test function explicitly to check that it works. Remove the call and run pytest on the module:

```
Terminal> py.test -s -v trapezoidal
```

Hint. Note that even if you know the value of the integral, you do not know the error in the approximation produced by the Trapezoidal rule. However, the Trapezoidal rule will integrate linear functions exactly (i.e., to machine precision). Base a test function on a linear $f(x)$.

c) Add functionality such that we can compute $\int_a^b f(x)dx$ by providing f, a, b, and n as positional command-line arguments to the module file:

```
Terminal> python trapezoidal.py 'sin(x)**2' 0 pi 20
```

Here, $a = 0$, $b = \pi$, and $n = 20$.

Note that the `trapezoidal.py` file must still be a valid module file, so the interpretation of command-line data and computation of the integral must be performed from calls in a test block.

Hint. To translate a string formula on the command line, like `sin(x)**2`, into a Python function, you can wrap a function declaration around the formula and run `exec` on the string to turn it into live Python code:

```
import math, sys
formula = sys.argv[1]
f_code = """
def f(x):
    return %s
""" % formula
exec(code, math.__dict__)
```

The result is the same as if we had hardcoded

```
from math import *

def f(x):
    return sin(x)**2
```

in the program. Note that `exec` needs the namespace `math.__dict__`, i.e., all names in the `math` module, such that it understands `sin` and other mathematical functions. Similarly, to allow a and b to be `math` expressions like `pi/4` and `exp(4)`, do

```
a = eval(sys.argv[2], math.__dict__)
b = eval(sys.argv[2], math.__dict__)
```

d) Write a test function for verifying the implementation of data reading from the command line.
Filename: `trapezoidal.py`.

Problem 3: Implement classes for the Trapezoidal rule

We consider the same problem setting as in Problem 2. Make a module with a class `Problem` representing the mathematical problem to be solved and a class `Solver` representing the solution method. The rest of the functionality of the module, including test functions and reading data from the command line, should be as in Problem 2. Filename: `trapezoidal_class.py`.

Problem 4: Write a doctest and a test function

Type in the following program:

```
import sys
# This sqrt(x) returns real if x>0 and complex if x<0
from numpy.lib.scimath import sqrt

def roots(a, b, c):
    """
    Return the roots of the quadratic polynomial
    p(x) = a*x**2 + b*x + c.

    The roots are real or complex objects.
    """
    q = b**2 - 4*a*c
    r1 = (-b + sqrt(q))/(2*a)
    r2 = (-b - sqrt(q))/(2*a)
    return r1, r2

a, b, c = [float(arg) for arg in sys.argv[1:]]
print roots(a, b, c)
```

a) Equip the `roots` function with a doctest. Make sure to test both real and complex roots. Write out numbers in the doctest with 14 digits or less.

b) Make a test function for the `roots` function. Perform the same mathematical tests as in a), but with different programming technology.
Filename: `test_roots.py`.

Exercise 5: Make use of a class implementation

Implement the `experiment_compare_dt` function from `decay.py` using class `Problem` and class `Solver` from Section 4. The parameters `I`, `a`, `T`, the scheme name, and a series of `dt` values should be read from the command line. Filename: `experiment_compare_dt_class.py`.

References

[1] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, fourth edition, 2014.

Index